Literature Review


In this paper the authors discuss about the technology which has been developed and implemented so that the software quality assurance activities can effectively and systematically be executed in software projects. Software quality assurance program provides guidelines and information concerning the three levels of SQA activities, the project SQA, the SQA management cycle, and the division SQA activity levels. Also the Meta-SQAP, project SQA activities, such as review, configuration control, and testing, can be well organized and planned out as a SQAP. Nearly all projects are supported by the SQA activities such as organizational restructuring and information exchange between projects.

C. Kramer and L. Prechelt (1996)

The authors introduce a concept of storing design pattern information in the form of C++ header files as a repository. PROLOG rules are formulated, which query the repository with the information extracted from the code, to identify a match. The authors successfully detect five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy, with a precision of 40 percent. Automatic detection of the design patterns instances probably the useful aid for maintenance purposes for quickly finding places where extensions and changes are most easily applied.


In this paper the authors state how to deal with the company standard requirements of SQA and also with the measurement activity required for the same purpose. It is very important to improve software quality using program analysis and measurement tools and the SQA (Software Quality Assurance) method at the appropriate points during the process of development. In many development departments, there is often not enough time to evaluate and use the tools and the SQA method or to accumulate the know-how for effective use. This paper describes the support
activity of a software quality analysis and measurement service which is performed by laboratory
team within the company as a third-party independent staff group.


In this paper the authors present the SPOOL (Spreading Desirable Properties into the Design of
Object-Oriented, Large Scale Software Systems) environment for the reverse engineering of
design components based on the structural descriptions of design patterns for C++ software. The
SPOOL environment consists of source code capturing tools, a design repository, and
functionality for pattern-based design recovery and representation. The project aims at both
software comprehension and software design quality assessment.


The authors discuss that SQUID (Software QUality In the Development Process) - a
measurement-based methodology for specifying, monitoring and evaluating the software product
quality during development - is adapted and used to manage the quality of implanting specific
tools, such as guides, checklists and templates for SQA and SCM practices. The SQUID
methodology is conceived and developed to manage the quality in the development of software
product. The authors also presented a proposal that allows adapting SQUID to be used as a tool
for the quantitative management of the quality of implanting generic tools.


This paper defines set of structural design patterns from OO design and the code, with higher
precision (55%), reduced false positives and improvised retrieval time, This is achieved with the
help of applying a multi-stage (Code and design) reduction strategy using software metrics and
structural properties. This approach does not take polymorphism into consideration.


The authors suggest a reverse engineered (code based) semi-automatic (with user intervention)
approach for pattern analysis. Code is put forth in the form of Abstract Syntax Graphs and
Patterns are described as graph transformations. In this approach, for the first time, design
patterns are formulated by a specification method; but each variant has to be described as a separate representation, which is quite cumbersome.

**Overmyer Scott, Lavoie Benoit, Rambow Owen (2001)**

This paper presents the fundamental problems faced by the software development community and its customers, in the early stages of software development is identification of objects, their attributes, and methods. These tasks are largely manual processes driven by heuristics that analysts or domain expert acquire through experience. The primary tools for object and model element identification and refinement are pencil and paper, and with the results being transferred to CASE tools after the analysis is largely completed.

This literature describes a methodology and a prototype tool called Linguistic assistant for Domain Analysis (LIDA), which provide linguistic assistance in the model development process. It presents a methodology to conceptual modeling through linguistic analysis. Then gives overview of LIDA's functionality and present its technical design and the functionality of its components. Finally, it presents an example of how LIDA is used in a conceptual modelling task.

**Kim and W. Shen (2002)**

In this paper the authors have advanced two separate approaches to design-level tool support, both based on the role-based metamodelling language RBML for specifying patterns. They translate RBML into a query on a Prolog database representing the UML model, but without explaining how to translate either language into Prolog. They report a plug-in called RBMLCC for IBM Rational Rose but then apply it to only 7 of the 23 GoF patterns.

**Mapdlsden, J. Hosking, and J. Grundy (2002)**

The authors proposes the Design Pattern Modelling Language (DPML), a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML supports incorporation of patterns at design-time, rather than program coding, assuming that if patterns can be effectively incorporated into a UML class model then conversion to code is straightforward. Though it is just a specification method no one has tested its effectiveness.

In this paper the authors address generating static and dynamic analysis algorithms from design pattern specifications to detect design patterns in legacy code. This is achieved with the help of representing legacy code by predicates that encode its attributed abstract syntax trees. Given these representations, the static analysis is performed on the legacy code representation as a query derived from the specification of the static pattern aspects which results in pattern candidates in the legacy code. The dynamic specification represents state sequences expected when using a pattern. They monitor the execution of the candidates and check their conformance to this expectation. The approach has identified Observer, Composite and Decorator patterns in Java code using Prolog to define predicates and queries. The limitation of this approach is that it requires phenomenal time, search space and generates false alarms. The above discussed code based design pattern detection approaches are suitable for reverse engineering, software comprehension and software maintenance purposes.


This paper proposes a Balanced Pattern Specification Language (BPSL) that uses both First Order Logic (FOL) and Temporal Logic of Actions (TLA) as formal basis in order to specify the structural as well as behavioural aspects of patterns. It focuses on how BPSL can be used to formally specify instances (specific solutions) of design patterns using the formulas of the original patterns with some substitutions. It has been experimented only with observer design pattern. It is rather complex way of specifying patterns, though it considers both structural and behavioural aspects.

Sergiu Dascalu, Ning Hao, Narayan Debnath (2005)

The authors have suggested an approach to detect design patterns with template library. Seven patterns have been implemented using C++ templates. This approach is better than GUI based Design Pattern Automation tools as it is open source and the user can get flexibility to modify pattern to suit his requirement. The downside is that, as compared to other wizard based tools, it has a demanding learning curve as the user has to learn template library before using it. The main limitation of all the above discussed approaches is that only structural facet of the design pattern
is recognized by code analysis; however the authors do not throw any light upon discovering dynamic aspects of the design patterns, through the methodologies such as message analysis.

**Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino (2005)**

The authors proposed a dual approach for pattern detection including code and design levels. At the code level, the input is OO source code, which is pre-processed by the Source Code Extractor to obtain an intermediate representation. At the design level, the Class Diagram Abstractor is able to import this representation to generate the corresponding graph structure. The SVG translator adds layout information to the graph, and the corresponding UML class diagram is translated in SVG format. This approach does not require a pre-processing phase to reduce the search space complexity and the cardinality of the set of the retrieved pattern candidates, as carried out by an earlier discussed method by Antonio. This tool lacks in scalability.

**Francesca Arcelli, Stefano Masiero, Claudia Raibulet (2005)**

In this paper the authors suggest a process of design pattern detection which is based on micro architectures recognition. Instead of analyzing source code directly, they summarizes it into a set of structures, called subcomponents or micro architectures, which are not ambiguous and involve a limited, number of types. This resolves the scalability issue. The results obtained are then processed by other two modules, one called Joiner that identifies sets of classes which could be “good" candidates to be design pattern instances and the other, called Neural Network which evaluates if the candidates identified by the Joiner are really “good" or not. Results obtained are not completely satisfactory but can be improved by building a larger and more balanced dataset, in order to have more reliable results. All these design based approaches cover structural aspects of design pattern. Later paragraph talk about the design based pattern recognition approaches which cover both structural and behavioural aspects of design patterns.

**Bayley and H. Zhu (2007)**

In this paper Bayley and H. Zhu has developed a tool, called LAMBDES-DP, with UML diagrams as the software design models. It facilitates model-driven development by working at
the design level rather than code level and it is based on a formalism that has been shown to be applicable to all 23 Gang of Four (GoF) patterns. This tool has a false positive rate of 22%.

Béla Újházi, Tibor Gyimóthy, Rudolf Ferenc and Denys Poshyvanyk (2009)

This paper presents 2 conceptual metrics which are measuring coupling and cohesion in software system.

1. Conceptual coupling between object classes (CCBO)-Based on coupling metric.
2. Conceptual lack of cohesion on methods (CLCOMS)- Based on cohesion metric.

Advantages of these metrics are that they can be evaluated or assessed as similar as structural metrics.

These metrics are used to predict the fault into the software system using the classes of the system and measured the metrics and compare these metrics with existing structural metrics.

This paper shows that coupling and cohesion metrics are used to measure and capture the degree of interaction and relationships among the source code elements such as classes, methods and attribute in object oriented (OO) software system.

Basically, this paper represents the main goals of OO analysis and design is to implement a software system where because classes have high cohesion and low coupling between the classes. Classes have these properties which gives facilitate activities, testing effects, reuse etc.

Y) Ju An Wang, HaoWang, MinzheGuo, and Min Xia (2009)

This paper proposes a new approach to define software security metrics based on vulnerabilities included in the software systems and their impacts on software quality. It is estimated that 90 per cent of reported security incidents result from exploits against defects in the design or code of software. To improve security there is need to measure it.

There are a great variety of different vulnerabilities existing for different kinds of software. Each has its own impact on quality and security attributes. Software security involves internal weakness and external attacks.

This approach is to select representative weaknesses that reflect the software security level. We use the Common Vulnerabilities and Exposures (CVE) lists to identify the weakness included in the software system during its lifecycle. Obviously more vulnerabilities discovered in a software
system would lead higher potential risks for the software system. In order to evaluate vulnerability, we need well-defined security metrics to measure the severity level of a vulnerability based on scientific, systematic, and quantitative approaches.

**Bandar Alshammari, Colin Fidge and Diane Corney (2009)**

This paper presents several metrics which have been developed for software quality attributes of object-oriented designs such as performance, reusability, and reliability. However, metrics which measure the quality attribute of information security have received little attention. Moreover, existing security metrics measure either the system from a high level (i.e. the whole system’s level) or from a low level (i.e. the program code’s level). These approaches make it hard and expensive to discover and fix vulnerabilities caused by software design errors. Here the input to the system will be UMLsec Diagram and the output will be the source code generated. The system accepts UMLsec Diagram and applies different refactoring methods to generate alternate designs. To identify most secure design it will apply the design level static metrics. Refactoring the class diagram to get multiple designs is the normalization of that class diagram. Refactoring is the process in which refactoring of the behaviour of classes is done to increase their cohesion and/or to reduce the coupling between them. This metric is defined to measure the weight of methods in a class which potentially interact with any classified attributes in a particular class. “The ratio of the number of classified methods to the total number of methods in a given class”.

**Martin Beck, Jonas Trumper and Jurgen Dollner Hasso (2011)**

In this paper Software Reengineering is not only involves adding new functionality or developing a completely new system based on an original system’s specification using forward engineering techniques.

Therefore, reengineering includes:

1. Reverse Engineering
2. Identify set of transformation
3. Apply these transformation
That’s why this paper represents visual analysis and design tool to support the planning for reengineering process of the existing software system. Basically tool can estimate an effort and change impact of a planning.

This paper represents two terms such as reengineering and refactoring which are the used into software development and maintenance. We can apply these terms to existing software system to modification its representation without changing its behaviour.

There are two types of refactoring;

1. Floss refactoring standing for small change into software system.
2. Root canal refactoring stands for large changes into software system.

The main goals of this paper are new concept for the visual analysis and design, thus it supports to evaluation of the as design as hierarchy and dependency and also identification of set of transformation.